Verification and Validation of High Integrity Software Generated by Automatic Code Generators*

Venkata Malepati, Hong Li, Krishna Pattipati
University of Connecticut, U-157, Dept. E.S.E.
Storrs, CT 06269-3157
phone/Fax: (860)486-2890/5585
Email: krishna@sol.uconn.edu

Ann Patterson-Hine
MS 269-4, NASA-Ames Research Center
Moffett Field, CA 94035-1000
Phone/Fax: (650)604-4178/4036
Email: apatterson-hine@mail.arc.nasa.gov

Abstract

In this paper, we present a comprehensive methodology for validating automatically generated software. The process consists of developing a fault-effect dependency model for the software, design of input stimuli to make the bugs manifest, design of tests to detect the incompatibilities from the intended behavior, and identifying the faulty code segment. A key feature of the methodology is the application of functional dependency modeling concepts, that are proven to be robust in the testability analysis and fault diagnosis of large complex hardware systems, to software verification and validation.

1 Introduction

The rapid advances in information technology have made software an indispensable element of daily life. The pervasiveness and increased complexity of software systems require that the software be produced efficiently and that its integrity and security be of the highest quality. Increasingly many systems and controllers are being implemented with the code generated by automatic code generators, such as MATRIXx SystemBuild [11] from Integrated Systems Inc. and SIMULINK [10] Real Time Workshop from the Mathworks, Inc. Before applying such auto-generated software confidently to safety-critical applications, techniques must be developed for assuring that the code is reliable.

Automatically generated software is cumbersome to debug manually. This is because it typically has more lines of code per module (unit), and smaller number of modules than manually generated code. In addition, all internal variables are not available for monitoring. This can mask many potential bugs from appearing at the system outputs. The McCabe's Cyclomatic complexity of individual modules [4], a measure of the complexity of a module's decision structure, is typically large. Consequently, it is difficult to manually test auto-generated code even at the unit-level.

*Research is supported by NASA - Ames Research Center under NASA-university consortium grant NCC2-5245

In this paper, we develop a model-based approach to validate the AutoCode generated by SystemBuild design environment, and a software tool implementing the software validation process. The software tool provides the following facilities:

1. Assessing software integrity via off-line testing, as well as real-time monitoring and troubleshooting; and
2. Evaluating software dependability and security via fault injection.

Our method consists of:

- Instrumenting the software so that bugs are detectable;
- Design of input stimuli via robust design techniques [5] so that bugs are "tickled";
- Design of tests so that wrong results are recognized ("error signatures");
- Real-time monitoring and troubleshooting of software via fast diagnostic inference algorithms to isolate faults quickly and accurately; and
- Assessing software integrity via fault injection.

Traditional software testing can be divided into two categories: static testing and dynamic testing [3]. Static testing involves requirement inspection and specification (model) testing, while dynamic (i.e., execution-based) testing includes black-box and glass-box approaches. Our process (shown in Figure 1) blends these two testing approaches. Consequently, it

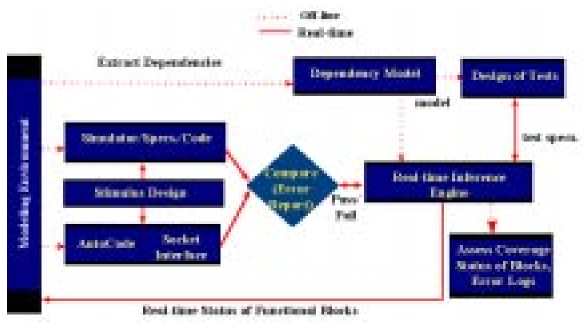


Figure 1: Overview of Monitoring and Validation of AutoCode

can be used for specification testing, software validation and troubleshooting. In the specification testing phase, our approach provides a priori predictions of detection and isolation measures and suggests suitable locations for test points (debug points). In the software validation and troubleshooting phase, faults are detected and isolated only with system-level input specifications.

In our process, the user provides a system model and input specifications for the software. Our software automatically extracts a dependency model of the system, where the modules are the functional blocks in the modeling environment (e.g., MATRIXx, SIMULINK). The I/O dependency model of each functional block is extracted via simulation. The dependency model of software can be used to design test points (i.e., variables to be monitored) for fault detection and isolation[1]. The inputs for system level testing are generated via robust design techniques based on system level input specifications (e.g., ranges of inputs). The ranges for input at unit-level are obtained via Monte-Carlo runs at system-level. The AutoCode results are compared against software model or its specifications (fault-free code in the case of software integrity assessment) to extract error signatures. The dependency of the software and the error signatures are used by the real-time inference engine[2] to assertain the status of functional blocks in software and to evaluate the integrity of software.

The paper is organized as follows. In Section 2, we discuss software instrumentation via TEAMS. In Section 3, we address the design of input stimuli to manifest bugs. In Section 4, we discuss the process of fault detection and error signature generation. In Section 5, we present a software monitoring and fault isolation process via TEAMS-RT. In Section 6, we demonstrate the application of the validation process to the Early External Active Thermal Control System (EEATCS) system. This is followed by Section 7 with a summary and suggested future enhancements.

[1]**TEAMS**, Testability Engineering And Maintenance System, an X-windows based testability engineering tool that generates testability figures of merits (TFOMS) from the fault-effect dependency models of systems, is used to design test points [2].

[2]**TEAMS-RT** , Testability Engineering And Maintenance System-Real Time, is used for real-time monitoring and fault isolation of the system using pass/fail test results [2].

2 Software Instrumentation via TEAMS

The goal of this step is to instrument the software so that bugs are detectable. The automatically generated software has limited assertions and self-checks that are not adequate for effective fault isolation. In order to instrument the software so that bugs are visible, it is necessary to extract a multisignal dependency model [1] of the software. Based on the multisignal model, the TEAMS software will identify the optimal set of "test points", i.e., positions in the code where additional tests and assertions need to be inserted (e.g., sanity checks on probabilities). This step is accomplished by automatically extracting a cause-effect dependency model from the function call hierarchy or from the design environment - such as CASE tools or via fault injection and simulation.

For the systems designed using MATRIXx, we extract a functional hierarchy, which defines potential dependencies, through System Build Accesses (SBA) - a querying tool to get the hierarchy, connectivity of the blocks, etc., in the system. The nature of dependency is extracted from the simulation of each functional block by injecting faults. All the available output variables of each functional block is a test in TEAMS. Another unique feature of TEAMS is that it can classify tests by run-levels (akin to debug levels used by developers) based on the computational requirements, depth and accuracy of a test, and evaluate the testability of software for different run-levels. The latter involves answers to questions, such as what percent of the software failures are detectable, where does a software failure propagate, the minimum set of variables to monitor for fault detection and isolation, etc.

Figures 2 shows a prototype software model of a subsystem of the space station, Early External Active Thermal Control System (EEATCS) in the MATRIXx design environment. The corresponding dependency model in TEAMS is depicted in Figure 3. The system has 13 inputs, 24 outputs, 19 superblocks with 5 levels of hierarchy. The AutoCode generated for this system has 1497 lines of code.

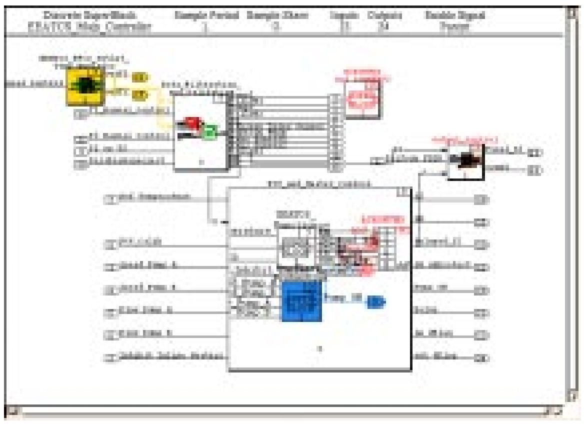


Figure 2: EEATCS System in SystemBuild

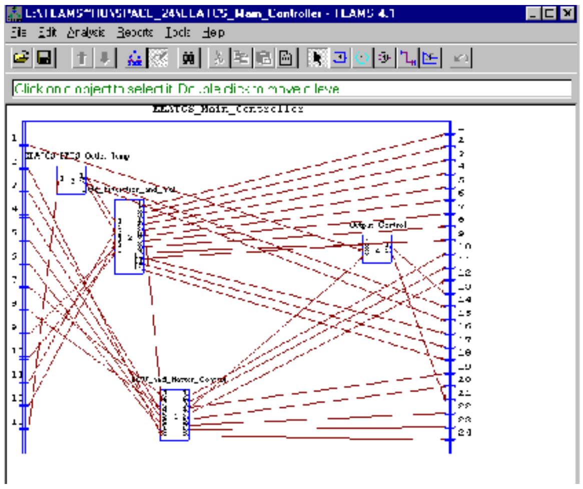

EEATCS_Main_Controller - TEAMS 4.1
File Edit Analysis Reports Tools Help
EEATCS_Main_Controller

Figure 3: Dependency Model of EEATCS in TEAMS

3 Design of Input Stimuli to Manifest Bugs

Based on the knowledge of the software specification, the input domain and possible levels of each input parameter are defined by well-known techniques, such as input partitioning, equivalence cases, and boundary value checking. For example, for each range (B1, B2) of a parameter, five sampled levels can be selected, corresponding to values less than B1, equal to B1, greater than B1 but less than B2, equal to B2 and greater than B2. Several design choices are provided in our software tool to construct test cases — *Orthogonal Arrays, One at a Time*, and *random testing*. In addition, users can define their own test cases or choose to apply all the test cases exhaustively.

One at a time test case generation uses a default normal condition as the starting point. The test cases are constructed by changing only one parameter at a time, assuming there is no interaction among parameters.

Random test case generation may be unpredictable for system-level testing. However Monte-Carlo simulation runs can be used to extract input information for unit-level blocks from the system-level input specifications. Thus, *Random* test case generation can be used as a prelude to unit-level testing.

Orthogonal Arrays is a fractional factorial design technique with following properties [6, 7]:

- The input levels are vertically balanced, and
- For any column, all levels occur equally.

The designs span over the input domain due to these properties. There are three resolution choices for Orthogonal Array generation: resolution III, IV and V. The higher the resolution, the more interaction effects are considered, typically requiring larger number of test cases with higher fault coverage.

A geometric view of *Orthogonal Arrays* versus *One at a Time* shows how Orthogonal Arrays provide better fault coverage. Figure 4 shows the case of three parameters and each parameter with three levels. If the test domain is divided into 64 unit cubes and if each test case can at most cover 8 unit cubes, the coverage for *Orthogonal Arrays* is 17/64, while that for *One at a Time* is only 10/64. In view of the superior fault coverage property, *Orthogonal Array* method is recommended.

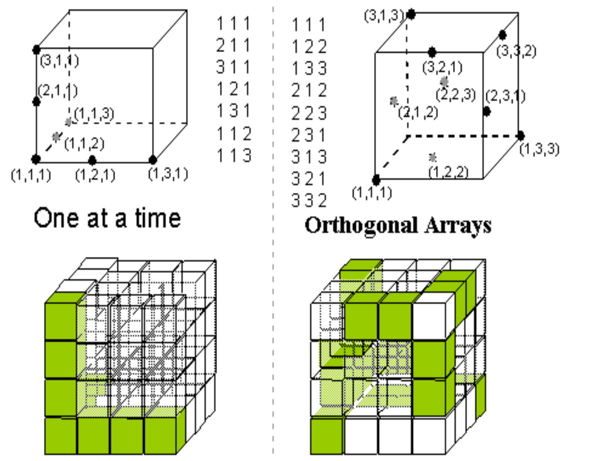
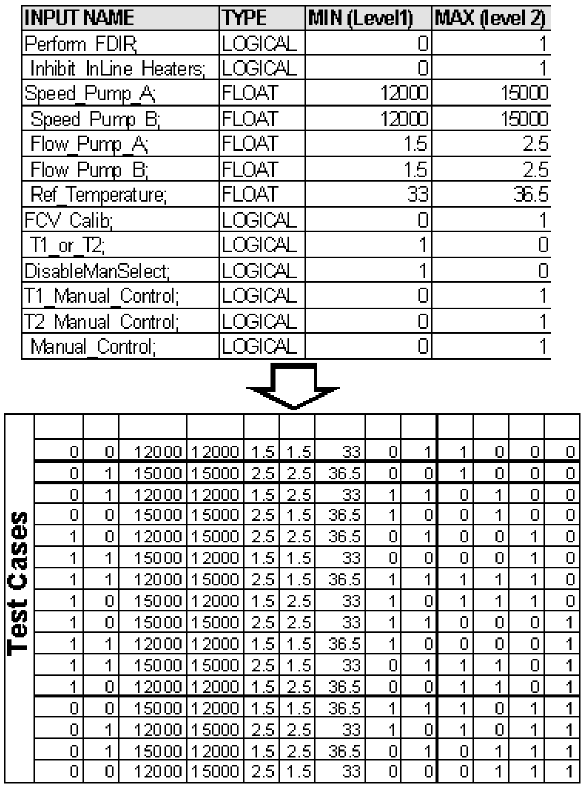


Figure 4: Orthogonal Arrays Vs. One at a Time

For the EEATCS, the input profile and test stimuli generated by way of *Orthogonal Arrays with resolution III* is illustrated in Figure 5. Here each input is partitioned into two levels and 16 test cases are constructed, while exhaustive testing would require

$2^{13} = 8192$ test cases. Exhaustive testing is infeasible due to combinatorial explosion. Consequently, test stimuli are designed to satisfy the following two objectives: Maximizing the chances of detecting a fault, while minimizing the number of test cases. With only 16 test cases from Orthogonal Arrays, faults are detected and two blocks are isolated to be faulty by system testing (see Section 4).



INPUT NAME	TYPE	MIN (Level1)	MAX (level 2)
Perform_FDIR;	LOGICAL	0	1
Inhibit_InLine_Heaters;	LOGICAL	0	1
Speed_Pump_A;	FLOAT	12000	15000
Speed_Pump_B;	FLOAT	12000	15000
Flow_Pump_A;	FLOAT	1.5	2.5
Flow_Pump_B;	FLOAT	1.5	2.5
Ref_Temperature;	FLOAT	33	36.5
FCV_Calib;	LOGICAL	0	1
T1_or_T2;	LOGICAL	1	0
DisableManSelect;	LOGICAL	1	0
T1_Manual_Control;	LOGICAL	0	1
T2_Manual_Control;	LOGICAL	0	1
Manual_Control;	LOGICAL	0	1

Test Cases

0	0	12000	12000	1.5	1.5	33	0	1	1	0	0	0
0	1	15000	15000	2.5	2.5	36.5	0	0	1	0	0	0
0	1	12000	12000	1.5	2.5	33	1	1	0	1	0	0
0	0	15000	15000	2.5	1.5	36.5	1	0	0	1	0	0
1	0	12000	15000	2.5	2.5	36.5	0	1	0	0	1	0
1	1	15000	12000	1.5	1.5	33	0	0	0	0	1	0
1	1	12000	15000	2.5	1.5	36.5	1	1	1	1	1	0
1	0	15000	12000	1.5	2.5	33	1	0	1	1	1	0
1	0	15000	15000	2.5	2.5	33	1	1	0	0	0	1
1	1	12000	12000	1.5	1.5	36.5	1	0	0	0	0	1
1	1	15000	15000	2.5	1.5	33	0	1	1	1	0	1
1	0	12000	12000	1.5	2.5	36.5	0	0	1	1	0	1
0	0	15000	12000	1.5	1.5	36.5	1	1	1	0	1	1
0	1	12000	15000	2.5	2.5	33	1	0	1	0	1	1
0	1	15000	12000	1.5	2.5	36.5	0	1	0	1	1	1
0	0	12000	15000	2.5	1.5	33	0	0	0	1	1	1

Figure 5: Input Stimuli for EEATCS

4 Fault Detection and Error Signature Generation

Fault detection and error signature generation involves the following steps:

- Apply stimuli and look for bugs (i.e., crash, hang, produce inaccurate results)

  This step is accomplished by employing input stimuli identified in *Section 3* to functionally exercise the system, and by monitoring the AutoCode variables identified in *Section 2*. We added a special preprocessing step in AutoCode to provide access to all the internal variables for monitoring.

- Design tests to recognize wrong results ("error signatures")

  The AutoCode is compiled and simulated for a user specified simulation time horizon. Depending on different user objectives, test results can be obtained by comparing the AutoCode simulation results with either output specifications or the results of design environment, i.e., SystemBuild simulator, or by comparing another version of Auto/manual code as shown in Figure 6. This assumes that the simulation results of design environment are validated prior to AutoCode validation.

- Generation of error signatures

  The error signatures can be computed in one of the following ways:

  1. At each time step
     $e_{ij}(t) = f[y^a_{ij}(t) - y^e_{ij}(t)]$;
     $i = 1, 2 \ldots NR; j = 1, 2 \ldots NM; 0 \le t \le t_f$
     where $y^a_{ij}(t)$ = actual value of the monitoring variable j in ith run at time t

     $y^e_{ij}(t)$ = expected value of the monitoring variable j in i th run at time t

     NR = number of runs (test cases)

     NM = number of monitoring variables

     t_f = simulation time horizon
  2. Aggregate over time for each run
     $e_{ij} = g[\{e_{ij}(t) : 0 \le t \le t_f\}]$;
     where g is typically a combined threshold function and a logical operator. For example,

     $$e_{ij} = \begin{cases} 1 & \text{if } e_{ij} > \Delta_{ij} \text{ for any } t \in (0, t_f) \\ 0 & \text{otherwise} \end{cases}$$

     where Δ_{ij} is a user specified threshold.
  3. Aggregate over time over all runs
     $e_j = h[\{e_{ij}(t) : 0 \le t \le t_f \;\; i = 1, 2 \ldots NR\}]$
     where h is typically a combined threshold function and a logical operator

5 Software Monitoring and Fault Isolation

The error signatures are mapped to pass/fail results of the test in the inference engine, TEASM-RT. TEAMS-RT is a testability tool with real-time monitoring capability and fast diagnostic inference algorithms [9]. TEAMS-RT uses the same dependency

TABLE 1: TFOMS for EEATCS at debug level 2

Isolate to Level	5	4	3	2	1
%FD	97.40	97.40	97.40	97.40	97.40
%FI	15.38	15.38	83.33	93.59	96.50
Ambiguity Group Size	7.98	5.15	1.27	1.03	1.01
# of FS	77	77	77	77	77
# of TPs Used	22	22	18	16	10

FS = Failure Sources, FD = Fault Detection,
FI = Fault Isolation, TP = Test Point,

TABLE 2: TFOMS for EEATCS at debug level 5

Isolate to Level	5	4	3	2	1
%FD	100.0	100.0	100.0	100.0	100.0
%FI	41.03	43.59	97.44	97.44	100.0
Ambiguity Group Size	5.80	3.65	1.02	1.02	1.00
#of FS	77	77	77	77	77
# of TPs Used	37	36	25	18	12

model that was extracted in section 2. Based on pass/fail results of the tests, TEAMS-RT quickly decides the overall health of the software i.e., faulty, suspected, or known good code segment. The isolation can be done to different levels of hierarchy.

TEAMS-RT results can be used to focus the unit testing to suspected blocks. We can further isolate the faults to program statements via off-line interactive simulation of the suspected units in a debug mode.

6 Application of Testing Process to EEATCS

Tables 1 and 2 show the Testability Figures Of Merit (TFOMS) measures obtained at debug levels 2 and 5, respectively, for the EEATCS system. We can see that the isolation is 100% upto isolation level 1. The average ambiguity group size is nearly 1 for isolation level upto 3. In addition, as shown in Table 2, one needs to monitor only 12 variables to detect a fault. On average, isolation can be done to one of five blocks at the lowest level of isolation (isolation level 5).

Figure 7 shows the error signatures for EEATCS triggered by the 16 test case orthogonal array discussed in Section 3. Two faulty blocks, *Set_S2_for_Delay* and *FCV Command Limit*, manifest errors and are isolated. Faulty statements in blocks are identified via interactive simulation. For the block *Set_S2_for_Delay* (Figure 8), when the last *elseif* is executed, *out_dFlag* is updated, but *delayed_S2* is not. In this case, the design environment resets *delayed_S2* to zero, while the auto-generated code keeps its previous value. The Block *FCV Command Limit* manifests a similar problem.

Another three blocks, *Moving_Average_T1*, *PFCS_Outlet_Temp_Seletionp*, and *Deadband Controller* did not manifest errors in those 16 test cases, but they also have similar initialization problems. These were identified during unit-level testing.

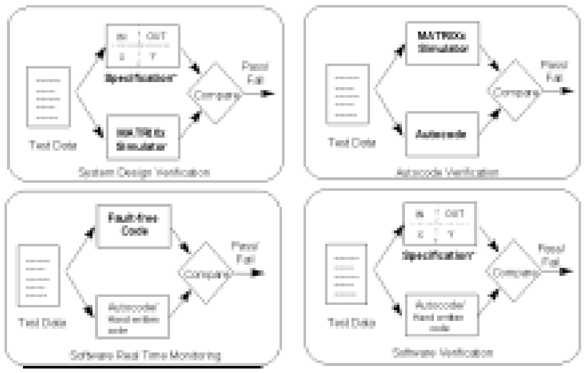
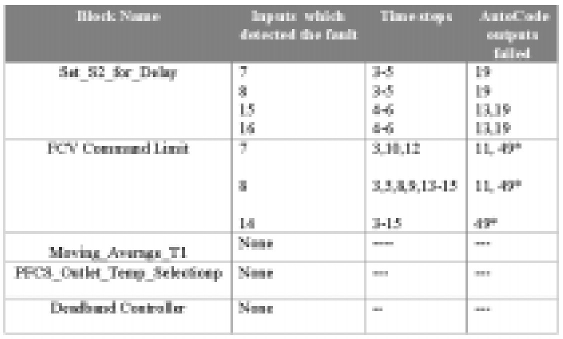


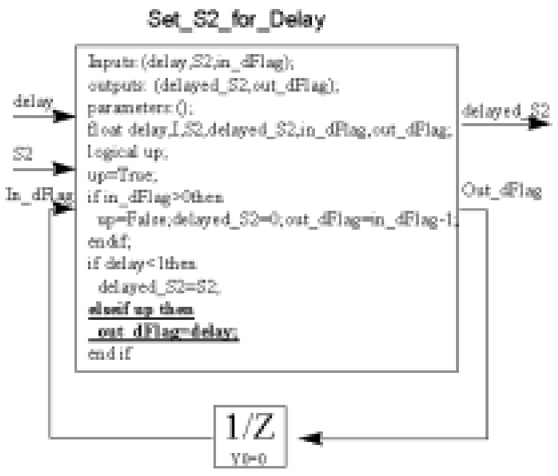
Figure 6: Different ways of Performing Verification and Validation



Block Name	Inputs which detected the fault	Time steps	AutoCode outputs failed
Set_S2_for_Delay	7 8 15 16	3-5 3-5 4-6 4-6	19 19 13,19 13,19
FCV Command Limit	7 8 16	3,10,12 3,5,8,9,13-15 3-15	11, 49* 11, 49* 49*
Moving_Average_T1	None	----	---
PFCS_Outlet_Temp_Selectionp	None	---	---
Deadband Controller	None	--	---

Figure 7: Error signatures for EEATCS

The discrepancy between the design environment and AutoCode causes errors. The discrepancy can be fixed via additional statements to initialize unupdated variables in the AutoCode or by making the SystemBuild and AutoCode to match by the design environment designer. The flaw is identified by the real-time inference engine, TEAMS-RT, in our software validation tool as shown in Fig 9. It would have



```
Inputs:(delay,S2,in_dFlag);
outputs: (delayed_S2,out_dFlag);
parameters:();
float delay,I,S2,delayed_S2,in_dFlag,out_dFlag;
logical up;
up=True;
if in_dFlag>0then
  up=False;delayed_S2=0;out_dFlag=in_dFlag-1;
endif;
if delay<1then
  delayed_S2=S2;
elseif up then
  out_dFlag=delay;
end if
```

Figure 8: Faulty Block *Set_S2_for_Delay*

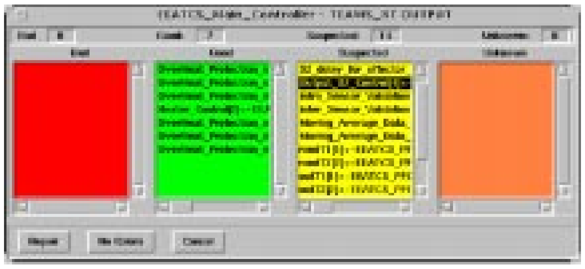


Figure 9: TEAMS-RT Output

been tedious and time-consuming to debug the entire code manually.

7 Summary

In this paper, we developed a process for verification and validation of automatically generated software by applying functional dependency modeling techniques. We also discussed efficient input stimuli generation, design of test points and application of real-time monitoring techniques to identify the faults quickly and accurately.

Application of the process to EEATCS system, a small subsystem of the space station, shows that automatic code generators need to be verified and validated. The process we developed for enhancing the quality of AutoCode helps engineers to design and evaluate software systems efficiently.

We are currently working on enhancements to make our methodology robust. The enhancements include other test case generation methods, sophisticated post-processing tests and error logging, application to other automatic code generators, and applications to general purpose software.

References

[1] Deb, S., Pattipati, K., Shakeri, and M., Shrestha, M., *"Multi-Signal Flow Graphs: A Novel Approach for System Testability Analysis and Fault Diagnosis,"* in Proc. IEEE AUTOTESTCON, Anaheim, CA, pp. 361-373, Sept. 1994.

[2] S. Deb *et al* "QSI's Integrated Diagnostic Toolset," *Proc. IEEE AUTOTESTCON,* Anaheim, CA, 1997.

[3] Schach, S.R, "Testing: Principles and Practice," Chapter 110 in *The Computer Science and Engineering Handbook*, Allen B. Tucker (Ed.), CRC Press, 1996.

[4] McCabe, T. J., "A Complexity Measure," *IEEE Trans. Software Eng.*, SE-2, pp. 308–320, 1976.

[5] Madhav S. Phadke, *"Quality Engineering Using Robust Design,"* Prentice Hall, Englewood Cliffs, N.J., 1989.

[6] Madhav S. Phadke, "Planning Efficient Software Tests," *The Journal of Defense Software Engineering*, pp. 11–15, Oct. 1997.

[7] Thomas P. Turiel, "A Computer Program to Determine Defining Contrasts and Factor Combinations for Two-Level Fractional Factorial Designs of Resolution III, IV, and V," *The Journal of Quality Technology*, pp. 267–271, Oct. 1988.

[8] Shakeri, M., Pattipati, K., Raghavan, V., Patterson-Hine, A., and Kell, T., "Sequential Test Strategies for Multiple Fault Isolation," in *Proc. IEEE AUTOTESTCON*, Atlanta, GA, Aug. 1995.

[9] Somnath Deb, Amit Mathur, Peter K. Willet, Krishna R. Pattipati., "De-centralized Real-time Monitoring and Diagnosis," in *Proc. IEEE Conference on System Man Cybernetics*, La Jolla, CA, Oct. 1998.

[10] *Simulink: Dynamic System Simulation Software, Release Notes, version 1.3*, The Mathworks, Inc., 1994, 1995.

[11] *System Build User's Guide for Version 5.0*, Integrated Systems, Inc., 1994, 1995.